DEVELOP

VIDEOGAME CREATION UNDER THE MICROSCOPE

Pushing the frontiers: extensibility in games

One common feature of many of the most successful and long-lived games is that they provide options for users to customise the game – adding new levels, monsters, sounds or simply tweaking the gameplay options. It is partly due to these features that games such as *Quake* and *Red Alert* are still heavily played today despite their relative age, with active players constantly releasing new patches for them. There are several ways, each with its own benefits and pitfalls, in which a developer can extend such options to build player loyalty.

The extensibility of ID Software's *Quake*, is provided by its built-in scripting language *Quake C*. This code is written to handle the motion and responses of every object in the game, so it's possible to change any of these behaviours simply by changing the code. *Quake C* is a very powerful language in many respects; patches such as the Reaper Bot, a computer opponent for multiplayer mode, demonstrate this well. *Quake C* also has the major benefit of being hardware-independent – patches written for one *Quake* version will work without modification on another. This approach has some drawbacks, however, the major one being speed. As *Quake C* is effectively an interpreted language, it can be slower than the equivalent native code, a factor that severely affects complex routines such as monster AI. *Quake C* is also confined to performing only those operations that the main *Quake* program provides, limiting its usefulness.

These factors prompted the switch in *Quake II* from *Quake C* to a DLL (Dynamic Link Library) system. Using a DLL means that the *Quake II* control routines can be programmed in normal *C*, *C++* or any other language which can compile to a DLL, and can access not only the functions provided by *Quake II*, but any other system functions they require. As DLLs are compiled in native code, they are also significantly faster than *Quake C* patches. The DLL approach does have its disadvantages: the portability of *Quake C* is lost, as is a degree of the protection offered by *Quake*'s 'sandboxing' of the *Quake C* code, which makes it impossible to affect the rest of the system adversely.

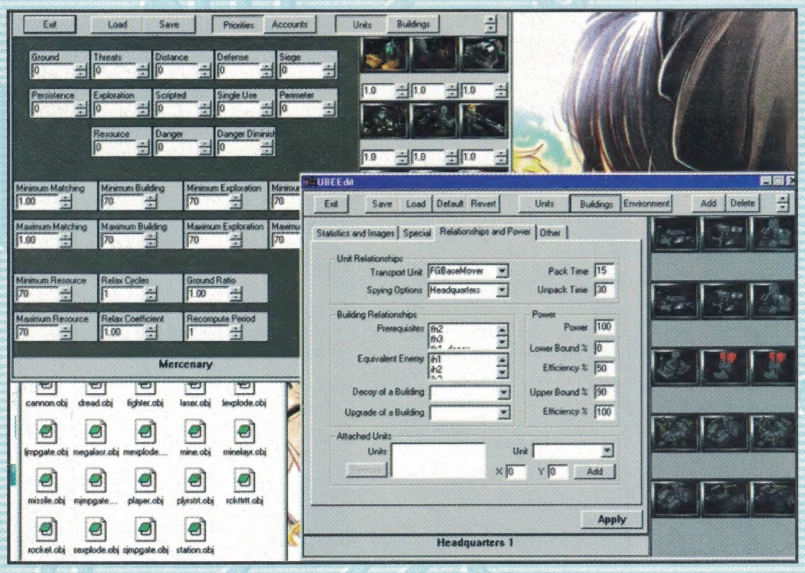
A simpler customisation option is to make available only the parameters used by the game logic, rather than the logic system itself. Many strategy games, such as *Red Alert*, have the controlling variables in a separate file (often a Windows-style .INI file, as these are easy to read and modify). A file may contain sections for each type of unit in the game, for example, along with its health rating, its maximum speed, its chosen sprite and so on. This makes it possible for users to change parameters with ease if they want to make a particular unit more or less powerful, or even to invent a completely new unit. It also aids the developer, as the game can be quickly modified during playtesting to try out new ideas and balance the gameplay.

Users may want to change the graphics, sound and music too. Here the developer needs to provide a converter that translates files between the game's own formats and other commonly-used formats, or even just to offer relevant documentation – if the game is popular, someone else will soon write a conversion program. Even a CD player option can dramatically improve things – everybody can have the in-game music they want.

A more recent and still evolving development is the shared universe concept, where several games based around the same world can be interconnected. A flight simulator and a tank game could be connected together, for example, with players able to drive or fly through the same environment. This can be made possible by using a unified representation for the world, and allowing each machine in the game to manipulate objects as required. The tank simulator may not know how an aircraft is supposed to move, but the flight simulator controlling the aircraft does. Alternatively, games could be linked on different levels, so that a strategy game could run the overall game, for example, calling on a simulator game only to fight the actual battles and pass the results back to it .

Some developers and publishers would argue that it is self-defeating to permit easy game modifications, as this enables other people to produce upgrades (free or otherwise) that compete with their own. This may be true in a few cases but the best upgrade packs rely on something different, either special changes to the game engine itself or the use of better-designed components, to set themselves apart from third-party offerings. The original developer is more likely than to understand how to keep the game balanced and stretch the engine.

Offering extensibility can only mean better games with more connectivity options and increased control for the user. What developers should appreciate, however, is that allowing the user to change the game options is not an excuse for failing to set them up correctly in the first place. E



Activision's *Dark Reign* provides extensive editing facilities for changing many aspects of the game; just one of the options that any enterprising games developer should consider

```
; Only the [Object] and [Sprite] sections are mandatory
; Note that all targetting-type distances are squared.  (is any distance used for target checks)

[Object]
Name=Player             ; The name of the object, as referred to in the game & in other object definitions

[Sprite]
Filename=plyr           ; The first 4 characters of the sprite filenames (<xxxx><num>.bmp/.ani, where <num>=0000-0360)
Sequential=No           ; Are the filenames sequential (0000,0001,0002), or angled?
TotalAngles=360         ; If not sequential, what is the highest angle used? (only useful for animation objects)
Animation=No            ; If set, then the object is assumed to be an animation, and spr0000+angle000,spr001+angle001,etc
Sprites=45              ; The number of sprites there are (1-360)
XSize=64                ; The size of the sprites
YSize=64
RadarColour=99999       ; The colour of the object on the radar

[Physics]
Speed=256               ; Actually the accelerations of the object
MaxSpeed=8192           ; The max speed of the object (missiles and other weapons are fired at max speed)
TurnSpeed=16            ; The turning speed of the object (negative turning speed indicates instantaneous turning)
TiltSpeed=300           ; The tilting speed of the object
BrakeNumerator=8        ; The braking "speed" (each frame, speed=(speed*BrakeNumerator)/BrakeDenominator
BrakeDenominator=11
Radius=28               ; The object radius (for collision detection)
Lifespan=0              ; The lifespan of the object. 0 indicates infinite lifespan
CollisionMask=4         ; Which object can it collide with? If (CollisionMask AND TargetClass)<>0 then collision occurs
UncontrollableJerk=No   ; If set to YES, then the player cannot override the object controls - useful for animations.

[Weapons]
PrimaryLaserBolt        ; Name of object to fire for primary weapon
PrimaryReloadDelay=2    ; Reload delay
PrimaryWeaponPods=1     ; Number of weapon pods on ship (0-4) (if 0, weapons can't fire!)
PrimaryWeaponPod1=-23   ; Weapon pod 1 x co-ord (taken directly off sprite. use PSP bottom-left-of-screen to find!)
PrimaryWeaponPod1Y=-19  ; Y1 y co-ord
PrimaryWeaponPod2=23    ; ... and so on ...
PrimaryWeaponPod2Y=-19

                        ; For secondary weapons, just duplicate the gunk above and put "Secondary" instead of "Primary"

Secondary=
SecondaryReloadDelay=2
SecondaryWeaponPods=1
```

Places to visit:

ID Software http://www.idsoftware.com/

Abuse (LISP-variant language) http://www.crack.com/games/abuse

Activision (Dark Reign editing and add-ons) http://www.activision.com/games/strategy/darkreign

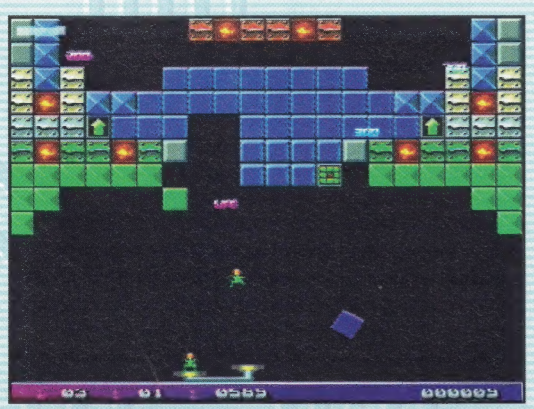
Yaroze: getting to grips with C

Apart from the £350 price tag (for a reduced period), the thing most likely to stop gamers buying the Yaroze PlayStation programming system and attempting to create their own PlayStation masterpieces is that simple phrase, 'Users will need a working knowledge of *C*'. Although most console owners are certain to know a little about the process of creating games, their knowledge usually stops at the meaning of terms like texture-mapping, prerendered scenery and realtime lighting. *C* itself, the programming language that makes it all possible, is likely to remain an enigma.

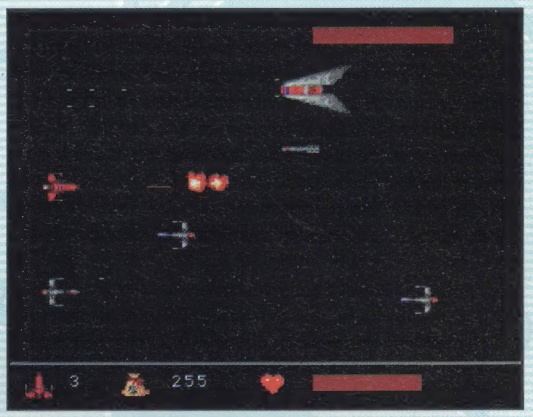
So what does a prospective Yaroze user need to know about *C*? Perhaps the first step is to realise that anyone who has used a console, PC or Mac over the last few years will have encountered it: *C* has quite simply become the language of choice for a huge number of programming tasks, not least games. Unfortunately, just one look at the way *C* is laid out – lots of '{' and '}' characters and alien commands – is enough to put many newcomers off. However, while it's not the most readable language in the world (it's very easy to write barely comprehensible *C*), it's not too dissimilar from a number of other programming languages.

In fact, knowledge of any other language will be a big help to the beginner learning *C*. Whether it's BASIC (preferably some form of structured BASIC like the revered BBC model), Pascal, Cobol, or even something like the STOS or AMOS construction kits, *C* is not a million miles away. What is important is an understanding of data structures, knowing the difference between BYTEs, WORDs and LONGs. *C* is essentially just another way of programming a computer, so whatever the user has done before with a programming language can be done in *C*. It is possible to buy a book and start learning the language from scratch, of course, but it's far better for users to try and replicate something they've already written in another language, so they can understand what the differences are.

Once even a basic grasp of *C* has been acquired, Yaroze users need to attempt doing something practical and visual with it. Unfortunately, it's not possible to just wade straight in and write *Ridge Racer* (despite what you might have heard); newcomers need to start with very basic routines to get them used to the hardware, and also to using *C*. Making sure that you can read joypad movement correctly is a good place to start; if users don't get that right, they're certainly not going to write a game in a hurry. Getting a simple shape up on the screen which responds to joypad prompts is the fundamental block on which to build.



Scott Evans' formula for *Bouncer 2* is a simple idea, but the game's programming will take months to master



George Bain's horizontally scrolling Yaroze shoot 'em up *StarFighter* may look basic, but the promising ideas it showcases were enough to secure him a job at SCEE

Be aware, however, that you won't get it right first time. Novice programmers often have to reorganise their code numerous times, because there will be something fundamental they have forgotten about. As Jonathan Cartwright, studio head at Corrosive Software asserts: 'It's a learning experience, and you shouldn't be downhearted if your early extravagant plans don't work out. Many of the Yaroze games are quite simple, and that's not a bad thing. If you're going to write a game, make sure you can achieve it. Even if you write some simple, stupid little demo, at least you've done it. And you can build on it from there. Nobody writes their best stuff straight away, and Yaroze is a limited format. However, its lack of memory and facilities should instill discipline, which will certainly pay off later...'

Talent Wanted

Several Net Yaroze members are looking for artists and musicians to collaborate on current game projects. Contact: ps_yaroze@playstation.sony.com

Competition news

For the last few months, Edge has been running a Yaroze programming competition with SCEE. The closing date for entries has now passed, and the winners will be announced in next month's issue.

Sign up

For information on how to become a Net Yaroze member (the only way to get hold of the machine), phone 0171 447 1616 or register online at www.scee.sony.co.uk

Alternatively write to:
Net Yaroze
SCEE
Waverly House
7-12 Noel Street
London W1V 4HH